MAC 110 — Introdução à Ciência da Computação

Aula 25

Nelson Lago

BMAC – 2024

Exercício — matriz

Escreva um programa que lê um arquivo de texto em que cada linha contém números separados por espaços e transforma o conteúdo desse arquivo em uma matriz

arquivo

```
1 2 3 4
5 6 7 8
9 10 11 12
```

Exercício — matriz

Escreva um programa que lê um arquivo de texto em que cada linha contém números separados por espaços e transforma o conteúdo desse arquivo em uma matriz

Exercício — matriz

Escreva um programa que lê um arquivo de texto em que cada linha contém números separados por espaços e transforma o conteúdo desse arquivo em uma matriz

```
def main():
```

Exercício — matriz

Escreva um programa que lê um arquivo de texto em que cada linha contém números separados por espaços e transforma o conteúdo desse arquivo em uma matriz

```
def main():
    nome_arq = input("Qual o arquivo? ")
    arq = open(nome_arq, "r")
```

Exercício — matriz

Escreva um programa que lê um arquivo de texto em que cada linha contém números separados por espaços e transforma o conteúdo desse arquivo em uma matriz

```
def main():
    nome_arq = input("Qual o arquivo? ")
    arq = open(nome_arq, "r")
    matriz = []
```

Exercício — matriz

Escreva um programa que lê um arquivo de texto em que cada linha contém números separados por espaços e transforma o conteúdo desse arquivo em uma matriz

```
def main():
    nome_arq = input("Qual o arquivo? ")
    arq = open(nome_arq, "r")
    matriz = []
    for linha in arq:
```

Exercício — matriz

Escreva um programa que lê um arquivo de texto em que cada linha contém números separados por espaços e transforma o conteúdo desse arquivo em uma matriz

```
def main():
    nome_arq = input("Qual o arquivo? ")
    arq = open(nome_arq, "r")
    matriz = []
    for linha in arq:
        l = linha.split()
```

Exercício — matriz

Escreva um programa que lê um arquivo de texto em que cada linha contém números separados por espaços e transforma o conteúdo desse arquivo em uma matriz

```
def main():
    nome_arq = input("Qual o arquivo? ")
    arq = open(nome_arq, "r")
    matriz = []
    for linha in arq:
        l = linha.split()
        for i in range(len(l)):
            l[i] = int(l[i])
```

Exercício — matriz

Escreva um programa que lê um arquivo de texto em que cada linha contém números separados por espaços e transforma o conteúdo desse arquivo em uma matriz

```
def main():
    nome_arq = input("Qual o arquivo? ")
    arq = open(nome_arq, "r")
    matriz = []
    for linha in arq:
        l = linha.split()
        for i in range(len(l)):
            l[i] = int(l[i])
        matriz.append(l)
```

Exercício — selection sort

Modifique nossa implementação anterior da função `selectionSort()` (“Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.”) para que todo o processamento seja feito em uma única função.

```
def selectionSort(L):
    for i in range(len(L)):
        certo = idxmenor(L, i)
        L[i], L[certo] = L[certo], L[i]

def idxmenor(L, início):
    menor = início
    for i in range(início, len(L)):
        if L[i] < L[menor]:
            menor = i
    return menor
```

Exercício — selection sort

Modifique nossa implementação anterior da função `selectionSort()` (“Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.”) para que todo o processamento seja feito em uma única função.

```
def selectionSort(L):
    for i in range(len(L)):
        menor = i
        for j in range(i, len(L)):
            if L[j] < L[menor]:
                menor = j
        L[i], L[menor] = L[menor], L[i]
```

Exercício — insertion sort

Modifique nossa implementação anterior da função `insertionSort()` ("Considera que o 'lado esquerdo' da lista está em ordem; a cada iteração, passa um elemento do lado direito para o lado esquerdo, garantindo que a ordem do lado esquerdo continue correta") para que todo o processamento seja feito em uma única função.

```
def insertionSort(L):
    for i in range(len(L)):
        encaixa(L, i)

def encaixa(L, i):
      while i > 0 and L[i-1] > L[i]:
        L[i-1], L[i] = L[i], L[i-1]
        i -= 1
```

Exercício — insertion sort

Modifique nossa implementação anterior da função `insertionSort()` (“Considera que o ‘lado esquerdo’ da lista está em ordem; a cada iteração, passa um elemento do lado direito para o lado esquerdo, garantindo que a ordem do lado esquerdo continue correta”) para que todo o processamento seja feito em uma única função.

```
def insertionSort(L):
    for i in range(len(L)):
        j = i
        while j > 0 and L[j-1] > L[j]:
            L[j-1], L[j] = L[j], L[j-1]
            j -= 1
```

Exercício — Módulos e Testes Automatizados

Suponha que você é parte de uma equipe que está desenvolvendo um site para venda online de produtos. Os arquitetos de software da equipe sugeriram a implementação das funções e módulos a seguir. Escreva uma boa bateria de testes automatizados para

- **manipulação do catálogo**
- **manipulação do estoque**
- **manipulação do carrinho**

Exercício — Módulos e Testes Automatizados

catalogo.py

```
acrescenta_produto_ao_catalogo(nome, código, preço)
    # devolve True caso tenha acrescentado com sucesso
    # devolve False se o preço é negativo
    # devolve False se o código do produto está repetido
remove_produto_do_catalogo(código)
    # devolve True caso tenha removido com sucesso
    # devolve False caso não tenha localizado o produto
lista_produtos_do_catalogo()
    # devolve uma lista de produtos, onde cada produto é uma tripla [nome, código, preço]
    # produtos são listados na ordem em que foram inseridos no catálogo
```

Exercício — Módulos e Testes Automatizados

estoque.py

```
adiciona_produto_ao_estoque(código, quantidade)
remove_produto_do_estoque(código)
    # devolve True caso tenha removido com sucesso
    # devolve False caso não tenha localizado o produto
quantidade_no_estoque(código_do_produto)
```

carrinho.py

```
adiciona_produto_ao_carrinho(código, quantidade)
    # devolve True caso tenha adicionado com sucesso
    # devolve False caso não tenha sucesso
remove_produto_do_carrinho(código, quantidade)
    # devolve True caso tenha removido com sucesso
    # devolve False caso não tenha sucesso
```

Exercício — Módulos e Testes Automatizados

```python
import pytest
import carrinho

@pytest.mark.parametrize("código, quantidade, result", [(2, 3, True),
                                                        (5, 0, False),
                                                        (5, 1, True)]
def test_carrinho1(código, quantidade, result):
    r = carrinho.adiciona_produto_ao_carrinho(código, quantidade)
    assert r == result
```

Exercício — Módulos e Testes Automatizados

```python
import pytest
import carrinho
@pytest.mark.parametrize("código, quantidade, result", [(1, 3, True), (1, 2, True),
                                                        (1, 0, False), (1, 5, True),
                                                        (1, 6, False), (7, 1, False),
                                                        (6, 3, True), (6, 4, False)]
def test_carrinho2(código, quantidade, result):
    carrinho.adiciona_produto_ao_carrinho(1, 5)
    carrinho.adiciona_produto_ao_carrinho(6, 1)
    carrinho.adiciona_produto_ao_carrinho(6, 1)
    carrinho.adiciona_produto_ao_carrinho(6, 1)
    carrinho.adiciona_produto_ao_carrinho(1, 0)

    r = carrinho.remove_produto_do_carrinho(código, quantidade)
    assert r == result
```

Divisão e conquista

Busca binária – divisão e conquista

Escreva uma função que recebe uma lista ordenada de números `L` e um número `n` e informa se o número está ou não na lista

```
def pertence(L, n):
    i = 0
    j = len(L) -1
    while i <= j:
        meio = (i + j) // 2
        if n == L[meio]:
            return True
        elif n > L[meio]:
            i = meio +1
        else:
            j = meio -1
    return False
```

Busca binária – divisão e conquista

Escreva uma função que recebe duas listas ordenadas de números `La`, `Lb` e um número `n` e informa se o número está ou não em uma delas

Busca binária – divisão e conquista

Escreva uma função que recebe duas listas ordenadas de números `La`, `Lb` e um número `n` e informa se o número está ou não em uma delas

```
def pertences(La, Lb, n):

```

Busca binária – divisão e conquista

Escreva uma função que recebe duas listas ordenadas de números `La`, `Lb` e um número `n` e informa se o número está ou não em uma delas

```
def pertences(La, Lb, n):
    return pertence(La, n) or pertence(Lb, n)
```

Busca binária – divisão e conquista

Mas uma lista com mais de um elemento é igual a duas sub-listas!

```
pertence(L, n)
```

$$\Leftrightarrow$$

```
pertence(L[:len(L)//2], n) or pertence(L[len(L)//2:]), n)
```

Busca binária – divisão e conquista

Escreva uma função que recebe uma lista ordenada de números `L` e um número `n` e informa se o número está ou não na lista

```
def pertence(L, n):
    if len(L) == 0:
        return False
    if len(L) == 1:
        return L[0] == n
    meio = len(L) // 2
    return pertence(L[:meio], n) or pertence(L[meio:], n)
```

Recursão

- **Recursão é útil quando você**

Recursão

- **Recursão é útil quando você**
  - Sabe como resolver o problema quando ele é “pequeno”

Recursão

- **Recursão é útil quando você**
  - Sabe como resolver o problema quando ele é “pequeno”
  - Sabe como dividir o problema “grande” em partes menores

Recursão

- **Recursão é útil quando você**
  - Sabe como resolver o problema quando ele é “pequeno”
  - Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*

Recursão

- **Recursão é útil quando você**
  - Sabe como resolver o problema quando ele é “pequeno”
  - Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*

Você sabe somar mais de dois números de uma vez?

Recursão

- **Recursão é útil quando você**
  - Sabe como resolver o problema quando ele é “pequeno”
  - Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*

Você sabe somar mais de dois números de uma vez?

$1 + 2 + 3 + 4 =$

Recursão

- **Recursão é útil quando você**
  - Sabe como resolver o problema quando ele é “pequeno”
  - Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*

Você sabe somar mais de dois números de uma vez?

$1 + 2 + 3 + 4 = (1 + 2) + 3 + 4 =$

Recursão

- **Recursão é útil quando você**
  - Sabe como resolver o problema quando ele é “pequeno”
  - Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*

Você sabe somar mais de dois números de uma vez?

$1 + 2 + 3 + 4 = (1 + 2) + 3 + 4 = 3 + 3 + 4 =$

Recursão

- **Recursão é útil quando você**
  - Sabe como resolver o problema quando ele é “pequeno”
  - Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*

Você sabe somar mais de dois números de uma vez?

$$1 + 2 + 3 + 4 = (1 + 2) + 3 + 4 = 3 + 3 + 4 = (3 + 3) + 4 =$$

Recursão

- **Recursão é útil quando você**
  - Sabe como resolver o problema quando ele é “pequeno”
  - Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*

Você sabe somar mais de dois números de uma vez?

$$1 + 2 + 3 + 4 = (1 + 2) + 3 + 4 = 3 + 3 + 4 = (3 + 3) + 4 = 6 + 4 =$$

Recursão

- **Recursão é útil quando você**
  - Sabe como resolver o problema quando ele é “pequeno”
  - Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*

Você sabe somar mais de dois números de uma vez?

$$1 + 2 + 3 + 4 = (1 + 2) + 3 + 4 = 3 + 3 + 4 = (3 + 3) + 4 = 6 + 4 = 10$$

Recursão

- **Recursão é útil quando você**
  - Sabe como resolver o problema quando ele é “pequeno”
  - Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*

Você sabe somar mais de dois números de uma vez?

$$1 + 2 + 3 + 4 = (1 + 2) + 3 + 4 = 3 + 3 + 4 = (3 + 3) + 4 = 6 + 4 = 10$$

- **Recursões são inspiradas nas demonstrações por indução da matemática**

Exercício – somatória

Escreva uma função recursiva que recebe uma lista de inteiros e devolve a soma dos elementos da lista

Exercício – somatória

Escreva uma função recursiva que recebe uma lista de inteiros e devolve a soma dos elementos da lista

```
def somatória(L):
```

Exercício – somatória

Escreva uma função recursiva que recebe uma lista de inteiros e devolve a soma dos elementos da lista

```
def somatória(L):
    if len(L) == 0:
        return 0
```

Exercício – somatória

Escreva uma função recursiva que recebe uma lista de inteiros e devolve a soma dos elementos da lista

```
def somatória(L):
    if len(L) == 0:
        return 0
    if len(L) == 1:
        return L[0]
```

Exercício – somatória

Escreva uma função recursiva que recebe uma lista de inteiros e devolve a soma dos elementos da lista

```
def somatória(L):
    if len(L) == 0:
        return 0
    if len(L) == 1:
        return L[0]
    return L[0] + somatória(L[1:])
```

Exercício – somatória

Escreva uma função recursiva que recebe uma lista de inteiros e devolve a soma dos elementos da lista

```
def somatória(L):
    if len(L) == 0:
        return 0
    if len(L) == 1:
        return L[0]
    return L[0] + somatória(L[1:])
```

Busca binária – divisão e conquista

Modifique a função `pertence()` para, ao invés de dividir a lista no meio, processar “o primeiro elemento” e “o resto da lista”.

Busca binária – divisão e conquista

Modifique a função `pertence()` para, ao invés de dividir a lista no meio, processar “o primeiro elemento” e “o resto da lista”.

```
def pertence(L, n):
    if len(L) == 0:
        return False
    if L[0] == n:
        return True
    return pertence(L[1:], n)
```

Recursão vs laços

- **Maneiras diferentes de pensar repetições**

Recursão vs laços

- Maneiras diferentes de pensar repetições
- Muitas vezes “dá na mesma”

Recursão vs laços

- Maneiras diferentes de pensar repetições
- Muitas vezes “dá na mesma”
- Às vezes, uma é mais simples de entender que a outra